UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

DIPARTIMENTO DI INFORMATICA

CORSO DI LAUREA IN INFORMATICA

TESI DI LAUREA
IN
SISTEMI AD AGENTI

Sviluppo e Gestione di Chatbot Addestrati con Tecnica RAG tramite Servizi RESTful

RELATORI:
Prof.ssa Berardina Nadja DE CAROLIS

LAUREANDO:
Federico Calò

ANNO ACCADEMICO 2023 - 2024

Indice

Capitolo 1

Introduzione

Nel contesto attuale, caratterizzato da una crescente interazione tra utenti e sistemi digitali, i chatbot stanno emergendo come strumenti essenziali per migliorare l'esperienza dell'utente e ottimizzare i processi aziendali. Diverse ricerche evidenziano come l'adozione di chatbot possa ridurre significativamente i costi operativi, aumentare l'efficienza del servizio clienti e migliorare il tasso di soddisfazione degli utenti. Ad esempio, uno studio condotto da Gartner nel 2023 stima che entro il 2026 i chatbot contribuiranno a ridurre i costi aziendali del 30

Tuttavia, i chatbot tradizionali presentano alcune limitazioni significative, in particolare la mancanza di contesto nelle risposte fornite. Questi sistemi spesso si basano su algoritmi deterministici o su modelli di machine learning addestrati su dataset limitati, che non sono in grado di adattarsi dinamicamente a richieste complesse o ambigue. Ad esempio, un chatbot tradizionale potrebbe fornire risposte errate o incomplete in scenari in cui l'utente utilizza un linguaggio non strutturato, si riferisce a informazioni specifiche o richiede dati aggiornati in tempo reale. Tali limitazioni riducono la loro efficacia in applicazioni aziendali critiche, come l'assistenza tecnica avanzata o la consulenza personalizzata.

Per superare queste limitazioni, è stata adottata la metodologia Retrieval-Augmented Generation (RAG), che combina tecniche di recupero di informazioni (retrieval) e generazione di risposte (generation). L'approccio RAG consente ai chatbot di consultare un database di conoscenze esterno per reperire informazioni pertinenti e

integrarle nella generazione di risposte, migliorando significativamente la pertinenza e la precisione delle interazioni. Ad esempio, un chatbot basato su RAG può accedere a documentazione tecnica, manuali aziendali o dati di mercato aggiornati, fornendo risposte dettagliate e contestualizzate anche a domande complesse.

Questa tesi si propone di affrontare la sfida della creazione e gestione di chatbot avanzati attraverso la progettazione e realizzazione di un ecosistema completo basato su RAG. L'obiettivo principale del progetto è sviluppare una piattaforma che consenta la creazione di diversi chatbot addestrati con questa metodologia, in grado di interfacciarsi tra loro e con l'utente attraverso servizi REST.

Per garantire una fruizione ottimale e una gestione efficiente dei chatbot, è stata progettata una soluzione integrata che comprende:

- **Frontend in ReactJS**: Un'interfaccia utente interattiva e dinamica che facilita l'interazione degli utenti con i chatbot. ReactJS è stato scelto per la sua capacità di creare applicazioni web veloci e reattive.
- **Servizi REST**: Interfacce di programmazione che permettono la comunicazione tra il frontend, i servizi backend e i vari bot. Le API sono state sviluppate utilizzando due tecnologie complementari: Spring Boot e Python.
- **Bot RAG**: L'approccio Retrieval-Augmented Generation è stato adottato per addestrare i chatbot, sviluppati in Python, combinando tecniche di recupero di informazioni e generazione di risposte, al fine di migliorare la pertinenza e l'accuratezza delle risposte fornite dai bot.

La realizzazione di questo ecosistema richiede un'integrazione armoniosa di componenti software e tecniche di machine learning, mirata a offrire una piattaforma versatile e potente per lo sviluppo di chatbot avanzati. L'importanza di questo progetto risiede nella sua capacità di semplificare la creazione e gestione di chatbot sofisticati, migliorando significativamente l'interazione uomo-macchina e offrendo soluzioni scalabili per diverse applicazioni aziendali e di consumo.

Diversi contesti possono beneficiare di quanto realizzato, tra cui il settore sanitario, che è stato scelto come esempio di applicazione pratica durante il progetto. In questo ambito, i chatbot basati su RAG possono supportare il personale medico

rispondendo a domande sui protocolli clinici, fornendo informazioni sui farmaci o assistendo i pazienti nella gestione delle loro cure.

Per quanto riguarda gli sviluppi futuri, questa tecnologia potrebbe evolversi integrando funzionalità di intelligenza artificiale più avanzate, come modelli di linguaggio di nuova generazione e algoritmi di apprendimento rinforzato. Ciò permetterebbe ai chatbot di adattarsi in tempo reale alle esigenze degli utenti, prevedere domande basandosi sul contesto e fornire un supporto ancora più personalizzato e proattivo. Inoltre, l'integrazione con sistemi IoT e dispositivi indossabili potrebbe ampliare ulteriormente le applicazioni, consentendo ai chatbot di raccogliere e analizzare dati in tempo reale per offrire risposte ancora più precise ed efficaci.

Capitolo 2

Large Language Models (LLM) e Retrieval-Augmented Generation (RAG)

L'intelligenza artificiale (IA) ha rivoluzionato il panorama tecnologico, con un impatto significativo su molteplici settori. Nel campo del Natural Language Processing (NLP), i Large Language Models (LLM) rappresentano una delle innovazioni più avanzate. Modelli come GPT (Generative Pre-trained Transformer) di OpenAI, BERT (Bidirectional Encoder Representations from Transformers) di Google e T5 (Text-To-Text Transfer Transformer) hanno dimostrato capacità straordinarie nel comprendere e generare linguaggio umano. Tuttavia, le loro limitazioni intrinseche, come la mancanza di accesso a dati aggiornati e il rischio di generare informazioni inesatte, ne riducono l'efficacia in applicazioni critiche.

Per superare queste sfide, è stata introdotta la tecnica Retrieval-Augmented Generation (RAG), sviluppata da Facebook AI Research (FAIR) nel 2020. RAG combina le capacità dei LLM con sistemi di recupero delle informazioni, consentendo l'accesso a fonti esterne per generare risposte più accurate e contestualizzate. Questo capitolo esplorerà il funzionamento degli LLM, le loro limitazioni e come RAG rappresenti un progresso cruciale per lo sviluppo di chatbot avanzati, supportato da studi e applicazioni pratiche.

Gli LLM sono modelli di deep learning progettati per comprendere e generare linguaggio naturale. Basati sull'architettura Transformer, introdotta da Vaswani et al. nel 2017 nel paper "Attention is All You Need", questi modelli sono addestrati su enormi corpus testuali che includono libri, articoli scientifici e conversazioni.

Un esempio emblematico è GPT-3, descritto nel paper "Language Models are Few-Shot Learners" di Brown et al. (2020). Con 175 miliardi di parametri, GPT-3 eccelle in compiti complessi come traduzioni, creazione di contenuti e scrittura di codice. Altri modelli di rilievo includono BERT, focalizzato sulla comprensione del contesto bidirezionale, e T5, che unifica diversi compiti NLP in un framework text-to-text.

Gli LLM sono utilizzati in numerosi ambiti:

- Generazione di testo: Creazione di articoli, riassunti e traduzioni.
- Chatbot: Assistenza virtuale e customer service.
- Analisi del linguaggio: Classificazione di testi e rilevamento del sentiment.
- Ricerca di informazioni: Risposte a domande complesse basate su grandi volumi di dati.

Nonostante il loro successo, gli LLM presentano limiti significativi che ne ostacolano l'applicazione in scenari knowledge-intensive. Uno dei problemi principali è la mancanza di accesso a informazioni aggiornate in tempo reale. Gli LLM generano risposte basate esclusivamente sui dati di addestramento, che potrebbero essere obsoleti. Questo è particolarmente problematico in domini come medicina e diritto, dove l'accuratezza e l'attualità delle informazioni sono cruciali.

Un altro limite è il fenomeno delle allucinazioni (hallucinations), dove i modelli generano risposte false o inventate. Ad esempio, se un utente richiede informazioni su un argomento emergente, un LLM potrebbe fornire dati inesatti. Inoltre, i bias presenti nei dati di addestramento possono influenzare negativamente le risposte, riducendo l'affidabilità del modello. La tecnica RAG è stata proposta da Lewis et al. nel 2020 nel paper "Retrieval-Augmented Generation for Knowledge-Intensive

NLP Tasks". Questo approccio ibrido combina la capacità di recupero delle informazioni con la generazione di testo, migliorando significativamente l'accuratezza e la pertinenza delle risposte.

Il funzionamento di RAG può essere spiegato in modo semplice e intuitivo. Immaginiamo di avere un assistente virtuale che deve rispondere a una domanda specifica, come "Quali sono le opzioni di spedizione disponibili?". Ecco come agisce il sistema RAG:

- Recupero: L'assistente cerca nei suoi archivi le informazioni più pertinenti alla domanda. Questa ricerca avviene utilizzando tecniche avanzate, come l'indicizzazione vettoriale (es. FAISS), che rendono il processo rapido ed efficiente.
- Fusione: Una volta trovate le informazioni, queste vengono "fuse" con la domanda dell'utente per creare un contesto ricco e dettagliato. È un po' come avere una conversazione con un esperto che mette insieme pezzi di conoscenza per darti una risposta completa.
- Generazione: Infine, il modello LLM interviene per trasformare tutto questo in una risposta ben articolata e naturale, che sembri davvero scritta da un essere umano.

L'approccio RAG offre diversi vantaggi significativi:

- Aggiornabilità: Poiché il sistema accede a database esterni, le informazioni sono sempre aggiornate, superando uno dei limiti principali degli LLM tradizionali.
- Riduzione delle allucinazioni: L'uso di fonti affidabili riduce drasticamente il rischio di risposte errate o inventate.
- Contestualizzazione: Le risposte non sono generiche ma adattate al contesto specifico della domanda, rendendole estremamente utili in settori come il customer service e l'educazione.

Torniamo all'esempio dell'assistente virtuale per il customer service. Con un sistema RAG, se un cliente chiede: "Quali sono le opzioni di spedizione disponibili?", il

chatbot recupererà le informazioni più recenti dal database aziendale, le integrerà con i dettagli della domanda e risponderà in modo chiaro e preciso: "Le opzioni disponibili sono: standard (3-5 giorni), espresso (1-2 giorni) e ritiro in negozio." Questo non solo migliora l'esperienza dell'utente ma aumenta anche la fiducia nei confronti del sistema. Guardando al futuro, RAG ha il potenziale per evolversi ulteriormente. Una possibile direzione è l'integrazione di modelli multimodali, che combinano testo, immagini e video per offrire risposte ancora più ricche e interattive. Pensiamo, ad esempio, a un chatbot che non solo risponde a una domanda tecnica ma mostra anche un video tutorial o un'infografica. Inoltre, l'adozione di algoritmi di apprendimento attivo potrebbe rendere il sistema più robusto, riducendo bias e migliorando la trasparenza.

Gli LLM hanno trasformato il panorama del NLP, ma le loro limitazioni richiedono approcci innovativi come RAG per superare le sfide attuali. La combinazione di generazione e recupero di informazioni consente di creare chatbot avanzati, capaci di offrire risposte accurate e contestualizzate. Studi come quelli di Lewis et al. (2020) e Izacard et al. (2021) evidenziano il potenziale di RAG in applicazioni knowledge-intensive. Con ulteriori sviluppi tecnologici, questa metodologia potrebbe rivoluzionare ulteriormente l'interazione uomo-macchina, aprendo nuove opportunità in settori strategici.

Capitolo 3

Intrroduzione ai LLMs e ai servizi REST (Da eliminare, è stato diviso in due)

I Large Language Models (LLMs) rappresentano una classe avanzata di modelli di intelligenza artificiale progettati per comprendere, generare e interagire con il linguaggio umano a livelli di complessità senza precedenti. Questi modelli sono stati all'avanguardia delle ricerche nel campo del Natural Language Processing (NLP) e sono stati progettati per comprendere, interpretare, generare e tradurre testi in linguaggio naturale. La loro "grandezza"(large) non deriva solo dalla dimensione del modello in termini di numero di parametri, ma anche dall'enrome quantità di dati su cui vengono addestrati. Questi modelli apprendono autonomamente le strutture linguistiche e le relazioni semantiche dai dati, permettendo loro di generare testi coerenti e pertinenti, rispondere a domande, riassumere documenti e molto altro.

I servizi RESTful (Representational State Transfer) rappresentano un'architettura per la creazione di applicazioni web e API che consente una comunicazione semplice ed efficiente tra client e server. Basati sui principi dell'architettura REST, questi servizi utilizzano il protocollo HTTP per gestire le richieste e le risposte tra le diverse componenti di un sistema, fornendo quindi un modo standardizzato e scalabile per la comunicazione tra queste parti. Essi giocano un ruolo cruciale nella connessione tra il frontend e il backend di sistemi complessi, come quelli che

utilizzano modelli di linguaggio avanzati. Per esempio, nel contesto di un chatbot addestrato con la metodologia RAG, i servizi RESTful possono facilitare la comunicazione tra l'interfaccia utente (frontend) e i servizi di elaborazione e generazione del linguaggio (backend), garantendo un'interazione fluida e reattiva.

Si vedrà in seguito come si sono combinate queste due tecniche di programmazione, il potenziale attuale e i futuri sviluppi.

3.1 Storia e caratteristiche degli LLMs

Il viaggio dei Large Language Models (LLMs) inizia negli anni '50, quando l'intelligenza artificiale era ancora agli albori. In quei primi giorni, i ricercatori si concentravano principalmente su sistemi basati su regole e approcci simbolici. I modelli di linguaggio dell'epoca erano piuttosto rudimentali e si basavano su semplici regole predefinite, senza la capacità di apprendere direttamente dai dati.

Con l'avvento del machine learning negli anni '80 e '90, le cose iniziarono a cambiare. I modelli statistici, come i modelli di Markov nascosti e gli n-grammi, iniziarono a essere utilizzati per l'elaborazione del linguaggio naturale (NLP). Sebbene questi modelli offrissero un miglioramento rispetto ai sistemi basati su regole, erano ancora limitati nella loro capacità di comprendere e generare linguaggio naturale in modo complesso.

La vera rivoluzione arrivò con l'introduzione delle reti neurali e del deep learning nei primi anni 2000. Architetture come le reti neurali ricorrenti (RNN) e le LSTM (Long Short-Term Memory) iniziarono a gestire meglio le sequenze di testo e le dipendenze a lungo termine, portando a progressi significativi nella NLP.

Ma è stato il 2017 a segnare una svolta decisiva con l'introduzione dell'architettura dei trasformatori nel paper "Attention is All You Need" di Vaswani et al. Questa innovazione ha rivoluzionato il campo, aprendo la strada a modelli come BERT (Bidirectional Encoder Representations from Transformers) e GPT (Generative Pre-trained Transformer). Questi modelli hanno dimostrato capacità straordinarie nella gestione del linguaggio, grazie alla loro abilità di catturare contesti complessi e gestire grandi quantità di dati.

Negli ultimi anni, i LLMs hanno raggiunto nuove vette con modelli di enormi dimensioni come GPT-3 e GPT-4. Questi modelli, con i loro miliardi di parametri, sono stati addestrati su enormi corpus di dati, permettendo loro di comprendere e generare testo con una precisione e coerenza senza precedenti.

Le caratteristiche distintive dei LLMs sono davvero notevoli. Innanzitutto, la loro grandezza è impressionante: modelli come GPT-3 hanno 175 miliardi di parametri, una quantità enorme che consente di catturare e rappresentare una vasta gamma di conoscenze linguistiche e semantiche. Questo grande numero di parametri è fondamentale per permettere ai modelli di comprendere e generare testo in modo accurato e fluido.

Inoltre, gli LLMs sono addestrati su enormi dataset provenienti da una varietà di fonti, tra cui libri, articoli e conversazioni online. Questo ampio addestramento consente ai modelli di apprendere un'ampia gamma di stili e contenuti linguistici.

Una delle caratteristiche più avanzate è la loro capacità di comprensione e generazione del linguaggio. Gli LLMs non solo comprendono il contesto delle conversazioni e dei testi, ma possono anche generare risposte pertinenti e coerenti, eseguire traduzioni, riassumere documenti e persino creare testi originali.

I LLMs utilizzano tecniche di apprendimento non supervisionato e pre-addestramento per costruire una base solida di conoscenze linguistiche. Questo approccio consente ai modelli di acquisire una comprensione generale del linguaggio prima di essere affinati su compiti specifici, migliorando ulteriormente la loro capacità di applicarsi a vari compiti di NLP.

Infine, l'architettura dei trasformatori, alla base della maggior parte degli LLMs moderni, è fondamentale. I trasformatori utilizzano meccanismi di attenzione per pesare l'importanza delle diverse parti del testo, gestendo in modo più efficace le dipendenze a lungo termine e migliorando la comprensione del contesto.

In sintesi, i Large Language Models hanno rappresentato una delle più grandi innovazioni nell'ambito dell'intelligenza artificiale. La loro evoluzione ha trasformato il modo in cui interagiamo con il linguaggio e ha aperto nuove possibilità per applicazioni in vari ambiti, dall'assistenza clienti alla generazione di contenuti creativi.

3.1.1 Applicazioni degli LLM

I Large Language Models (LLMs) hanno trasformato radicalmente il campo dell'intelligenza artificiale e dell'elaborazione del linguaggio naturale (NLP), aprendo nuove possibilità applicative in vari settori. Questi modelli, grazie alla loro capacità di comprendere e generare testo con una precisione senza precedenti, sono diventati strumenti essenziali per una vasta gamma di applicazioni.

Una delle principali applicazioni degli LLMs è nei sistemi di assistenza virtuale. Tecnologie come Siri, Alexa e Google Assistant utilizzano LLMs per comprendere e rispondere alle richieste degli utenti, eseguire comandi vocali, fornire informazioni in tempo reale e interagire con altre applicazioni. Questi assistenti virtuali si basano sulla capacità dei LLMs di interpretare il linguaggio naturale, comprendere il contesto e generare risposte appropriate in modo fluido e naturale.

Nel settore della generazione di contenuti, i LLMs vengono utilizzati per creare testi automaticamente. Ad esempio, possono generare articoli di notizie, post sui social media, descrizioni di prodotti, e persino testi creativi come poesie o storie. Strumenti come GPT-3 hanno dimostrato di poter produrre contenuti di alta qualità che sono quasi indistinguibili da quelli scritti da esseri umani, accelerando il processo di creazione di contenuti e rendendolo più accessibile.

Gli LLMs sono anche ampiamente utilizzati nei sistemi di traduzione automatica. Modelli come quelli implementati da Google Translate e altri servizi di traduzione online sono in grado di tradurre testi da una lingua all'altra con una precisione e fluidità sempre maggiori. Questi modelli sfruttano la loro comprensione approfondita delle strutture linguistiche e delle relazioni semantiche per produrre traduzioni che rispettano il contesto e il significato originale.

Nel campo della analisi del sentiment e del monitoraggio dei social media, i LLMs vengono utilizzati per analizzare grandi volumi di dati testuali, come post sui social media, recensioni dei clienti e articoli di opinione, per comprendere l'opinione pubblica e il sentiment generale riguardo a prodotti, marchi o eventi. Questa analisi permette alle aziende di monitorare la percezione del pubblico e di prendere decisioni informate basate sui feedback raccolti.

Un'altra applicazione emergente degli LLMs è nel supporto alla scrittura e alla

ricerca. Strumenti come quelli sviluppati da OpenAI vengono utilizzati per assistere scrittori, ricercatori e studenti nel creare bozze, riassumere articoli, generare idee o formulare risposte a domande complesse. Questi modelli possono accelerare il processo creativo e migliorare la produttività, fornendo suggerimenti pertinenti e contenuti coerenti in tempo reale.

Infine, gli LLMs trovano applicazione anche nel supporto decisionale aziendale. Attraverso l'analisi di dati testuali e documenti, questi modelli possono aiutare le aziende a estrarre informazioni critiche, fare previsioni basate su trend emergenti e supportare la presa di decisioni strategiche. Questo è particolarmente utile in contesti come l'analisi di mercato, la gestione del rischio e la pianificazione aziendale.

3.2 Storia e caratteristiche dei servizi REST

Nel panorama attuale dello sviluppo software, i servizi RESTful hanno acquisito un ruolo fondamentale grazie alla loro semplicità ed efficacia nel facilitare la comunicazione tra diverse applicazioni e componenti di un sistema. L'architettura REST, acronimo di Representational State Transfer, è stata introdotta da Roy Fielding nel 2000 e ha rapidamente rivoluzionato il modo in cui le applicazioni web sono progettate e sviluppate.

Un aspetto chiave dei servizi RESTful è la loro natura stateless. Questo significa che ogni richiesta effettuata dal client verso il server è indipendente dalle altre; il server non conserva informazioni sullo stato delle richieste precedenti. Questa caratteristica semplifica notevolmente la scalabilità delle applicazioni, poiché non è necessario che il server gestisca sessioni o mantenga traccia delle interazioni passate. Ogni richiesta contiene tutte le informazioni necessarie per essere processata in modo autonomo.

La separazione tra client e server è un altro principio cardine di REST. In questa architettura, il client è responsabile della presentazione e dell'interfaccia utente, mentre il server gestisce la logica di business e l'archiviazione dei dati. Questo tipo di suddivisione permette uno sviluppo più modulare, in cui le componenti del frontend e del backend possono evolvere in modo indipendente. Ad esempio,

è possibile aggiornare l'interfaccia utente senza dover necessariamente intervenire sulla logica del server, e viceversa.

Un'altra caratteristica distintiva dei servizi RESTful è l'utilizzo di un'interfaccia uniforme per la comunicazione. I metodi standard del protocollo HTTP, come GET, POST, PUT e DELETE, sono utilizzati per eseguire operazioni su risorse identificate da URL specifici. Questo approccio non solo semplifica lo sviluppo, ma rende anche le API RESTful facili da comprendere e utilizzare, poiché seguono convenzioni ben definite e universalmente riconosciute.

Le risorse, in un'architettura RESTful, sono rappresentate tramite URL. Ogni risorsa, che può essere un oggetto, un servizio, o una qualsiasi entità gestita dal server, è accessibile tramite un URL univoco. Questo rende l'accesso e la manipolazione delle risorse intuitivi e facilmente gestibili, poiché tutto è strutturato attorno a un sistema di indirizzamento chiaro e organizzato.

Un ulteriore vantaggio offerto dai servizi RESTful è la possibilità di caching. Le risposte alle richieste possono essere memorizzate dal client per un certo periodo di tempo, riducendo così la necessità di ripetere richieste identiche e migliorando l'efficienza complessiva del sistema. Il caching permette di alleviare il carico sui server e di velocizzare l'accesso ai dati, migliorando l'esperienza utente.

Infine, l'architettura REST supporta la realizzazione di sistemi stratificati. Ciò significa che tra il client e il server possono essere introdotti vari componenti intermediari, come proxy e gateway, che svolgono funzioni aggiuntive come la gestione della sicurezza, il bilanciamento del carico e l'ottimizzazione delle prestazioni. Questa stratificazione permette di scalare il sistema in modo più efficiente e di aggiungere nuove funzionalità senza interrompere l'intera infrastruttura.

Nel complesso, i servizi RESTful hanno rivoluzionato il modo in cui le applicazioni web e mobile interagiscono con il backend. La loro semplicità, combinata con una grande flessibilità, li rende ideali per la costruzione di sistemi complessi e distribuiti, come i microservizi, che possono essere sviluppati, testati e aggiornati indipendentemente. La diffusione di questa architettura ha consentito lo sviluppo di applicazioni altamente scalabili e manutenibili, in grado di adattarsi rapidamente alle esigenze in continua evoluzione del mondo digitale.

3.2.1 Applicazioni dei servizi REST

I servizi RESTful hanno trovato un'ampia applicazione in numerosi settori e tipologie di sistemi, grazie alla loro flessibilità, semplicità e capacità di facilitare l'integrazione tra diverse componenti software. L'approccio REST è diventato lo standard de facto per la progettazione di API (Application Programming Interface), utilizzate per collegare diverse applicazioni, servizi e dispositivi in modo efficiente e scalabile.

Una delle applicazioni più comuni dei servizi REST è nello sviluppo di applicazioni web e mobile. Le API RESTful consentono a queste applicazioni di comunicare con il backend per recuperare dati, inviare informazioni e interagire con altri servizi. Ad esempio, un'app mobile di social media utilizza REST per caricare i feed degli utenti, pubblicare nuovi post, gestire notifiche e interagire con altre funzionalità della piattaforma. La natura stateless di REST rende possibile gestire un elevato numero di richieste simultanee, rendendo le applicazioni più rapide e reattive.

I servizi RESTful sono anche ampiamente utilizzati nei microservizi, un'architettura che scompone le applicazioni in una serie di piccoli servizi indipendenti, ciascuno dei quali svolge una funzione specifica. Questa struttura permette di sviluppare, distribuire e scalare i componenti in modo indipendente, riducendo la complessità e migliorando l'agilità del sistema. In questo contesto, REST funge da collante che permette ai vari microservizi di comunicare tra loro in modo standardizzato e coerente.

Nel campo dell'Internet of Things (IoT), REST viene spesso impiegato per facilitare la comunicazione tra dispositivi intelligenti e i sistemi backend che li gestiscono. Ad esempio, i dispositivi IoT, come termostati intelligenti o sistemi di monitoraggio remoto, utilizzano API RESTful per inviare dati sensoriali a un server centrale, che li elabora e restituisce comandi o configurazioni aggiornate. La leggerezza e la scalabilità di REST lo rendono ideale per gestire reti di dispositivi distribuiti e a bassa potenza.

Anche nel settore delle integrazioni tra sistemi eterogenei, i servizi RESTful sono ampiamente utilizzati. Grazie alla loro natura universale e indipendente dalla piattaforma, REST permette a sistemi sviluppati con tecnologie diverse di comunicare

tra loro. Ad esempio, un sistema di gestione delle risorse umane potrebbe utilizzare REST per sincronizzare dati con un sistema di contabilità o con applicazioni di terze parti per la gestione dei salari.

Infine, REST è largamente impiegato nella creazione di applicazioni cloud. I servizi cloud come AWS, Google Cloud e Microsoft Azure offrono API RESTful per gestire risorse cloud, come server, database e servizi di storage. Queste API permettono agli sviluppatori di automatizzare la gestione delle risorse cloud, implementando funzioni di scalabilità automatica, monitoraggio delle performance e gestione delle configurazioni.

3.3 Servizi REST e LLMs

L'interazione tra servizi RESTful e Large Language Models (LLMs) rappresenta un approccio potente e versatile per sfruttare le capacità avanzate dell'intelligenza artificiale in applicazioni web e mobile. I servizi REST, grazie alla loro natura semplice e standardizzata, forniscono un'infrastruttura ideale per orchestrare e gestire la comunicazione tra i client e i modelli di intelligenza artificiale, come gli LLMs, che sono in grado di comprendere e generare linguaggio naturale.

In un'architettura in cui i servizi RESTful fungono da intermediari tra l'interfaccia utente e i LLMs, ogni richiesta effettuata dal client può essere facilmente instradata verso il modello AI appropriato per l'elaborazione. Ad esempio, un utente potrebbe inviare una domanda o un comando tramite un'applicazione web o mobile. Questa richiesta viene trasmessa tramite una chiamata API RESTful al server, dove i servizi REST gestiscono la richiesta e la inoltrano all'LLM per l'elaborazione.

Uno dei principali vantaggi dell'utilizzo di servizi REST in combinazione con LLMs è la loro capacità di gestire richieste stateless. Questo significa che ogni richiesta inviata al server è indipendente e contiene tutte le informazioni necessarie per essere elaborata. Gli LLMs, che possono richiedere risorse computazionali significative, possono così concentrarsi sull'elaborazione del linguaggio naturale senza doversi preoccupare dello stato della sessione utente. Questo approccio semplifica l'infrastruttura e migliora la scalabilità del sistema.

Un'altra area in cui i servizi REST possono migliorare l'integrazione con gli LLMs

è attraverso la modularità. I servizi RESTful possono essere progettati per svolgere funzioni specifiche, come l'autenticazione, la gestione delle richieste, o l'elaborazione di dati prima o dopo l'intervento degli LLMs. Questa modularità consente di creare pipeline di elaborazione flessibili, in cui diverse componenti possono essere aggiornate o sostituite senza interrompere l'intero sistema. Per esempio, è possibile implementare un servizio REST che pre-processa i dati dell'utente, rendendoli più facilmente comprensibili per l'LLM, migliorando così la precisione delle risposte generate.

Inoltre, i servizi RESTful possono facilitare l'integrazione di LLMs con altre risorse e servizi. Ad esempio, un'API REST può essere utilizzata per recuperare dati da un database o da una fonte esterna, che poi vengono combinati con le capacità generative dell'LLM per fornire risposte più complete e contestuali. Questo approccio, noto come Retrieval-Augmented Generation (RAG), permette agli LLMs di accedere a informazioni specifiche e aggiornate, migliorando significativamente la qualità delle risposte.

Infine, i servizi RESTful sono particolarmente adatti per automatizzare e scalare l'uso degli LLMs in applicazioni distribuite. Grazie alla loro compatibilità con il protocollo HTTP e la loro facilità di implementazione, i servizi REST possono essere utilizzati per orchestrare l'elaborazione di grandi volumi di richieste simultanee, bilanciando il carico tra diversi istanze dell'LLM e garantendo che le risposte siano fornite in tempi rapidi. Questo è cruciale per applicazioni come i chatbot su larga scala, dove l'interazione in tempo reale con gli utenti è essenziale.

In sintesi, l'integrazione di LLMs con servizi RESTful offre una soluzione potente e flessibile per implementare capacità avanzate di elaborazione del linguaggio naturale in applicazioni moderne. Grazie alla loro modularità, scalabilità e facilità di integrazione, i servizi RESTful non solo facilitano la comunicazione con i modelli di intelligenza artificiale, ma permettono anche di sfruttare appieno le loro potenzialità in un'ampia gamma di scenari applicativi.

Capitolo 4

Analisi e progettazione

Negli ultimi due decenni, i servizi RESTful (Representational State Transfer) hanno trasformato il modo in cui le applicazioni comunicano, emergendo come lo standard de facto per la progettazione di API scalabili e interoperabili. L'architettura REST, introdotta da Roy Fielding nella sua tesi di dottorato nel 2000, si basa su principi che enfatizzano la semplicità, l'indipendenza tecnologica e l'uso efficiente delle risorse di rete. La sua adozione su vasta scala ha facilitato l'interoperabilità tra sistemi eterogenei, rendendola un pilastro fondamentale dello sviluppo software moderno.

Un aspetto chiave del successo di REST è la sua natura stateless, che elimina la necessità per il server di mantenere lo stato delle richieste precedenti. Ogni richiesta HTTP include tutte le informazioni necessarie per essere processata, garantendo una maggiore scalabilità e semplificando l'architettura. Questo approccio consente ai servizi RESTful di gestire un elevato volume di richieste simultanee, un requisito essenziale per le moderne applicazioni distribuite.

La separazione tra client e server è un altro principio cardine di REST. Questo modello architettonico consente uno sviluppo modulare, in cui il frontend e il backend possono evolversi indipendentemente. Ad esempio, un'applicazione mobile può utilizzare le stesse API RESTful di un'applicazione web, garantendo una maggiore riusabilità del codice e riducendo i costi di manutenzione.

La storia dei servizi RESTful è strettamente legata alla crescita di Internet e delle applicazioni web. Negli anni '90, la necessità di una comunicazione efficiente tra sistemi in rete ha portato allo sviluppo di vari protocolli e architetture, ma molti

di essi, come SOAP (Simple Object Access Protocol), risultavano troppo complessi o rigidi per soddisfare le esigenze in rapida evoluzione delle applicazioni moderne.

L'introduzione di REST ha segnato un cambiamento radicale, grazie alla sua enfasi sulla semplicità e sull'utilizzo di standard esistenti come HTTP. La sua adozione è stata accelerata dall'ascesa di applicazioni basate su microservizi, dove l'indipendenza e la modularità dei componenti erano requisiti fondamentali. Studi recenti, come quello di Richardson e Ruby nel libro "RESTful Web Services", sottolineano come REST abbia permesso di superare le limitazioni delle architetture precedenti, offrendo una soluzione flessibile e scalabile per la progettazione di API.

Con l'emergere dei modelli di linguaggio di grandi dimensioni (LLMs), come GPT e BERT, i servizi RESTful hanno trovato una nuova dimensione di applicazione. L'integrazione tra REST e LLMs consente di sfruttare le capacità avanzate di elaborazione del linguaggio naturale all'interno di architetture distribuite, creando un'infrastruttura potente e flessibile per applicazioni basate sull'intelligenza artificiale.

In questo progetto, i servizi RESTful fungono da intermediari tra i modelli LLMs, i database e le applicazioni client. La tecnica Retrieval-Augmented Generation (RAG), utilizzata per arricchire le risposte dei chatbot con dati specifici e aggiornati, si basa su questo approccio. Ad esempio, una richiesta HTTP proveniente da un client può attivare il recupero di informazioni da un database, che vengono poi elaborate da un modello LLM per generare una risposta contestualizzata e pertinente.

4.1 Architettura del Sistema

Il progetto adotta un'architettura basata su microservizi, implementata con Spring Boot per il backend e Python per l'addestramento e l'interrogazione dei modelli di linguaggio. Questa scelta tecnologica consente di sfruttare i punti di forza di ciascuna piattaforma: Spring Boot per la robustezza e la scalabilità delle API REST, e Python per la sua vasta gamma di librerie dedicate al machine learning e al Natural Language Processing.

Ogni microservizio è progettato per svolgere una funzione specifica, come l'autenticazione, la gestione delle richieste o l'elaborazione dei dati. Questo approccio modulare facilita l'estensibilità del sistema, permettendo di aggiungere nuovi chatbot o funzionalità senza compromettere l'infrastruttura esistente. I servizi RESTful, grazie alla loro natura indipendente dalla piattaforma, orchestrano la comunicazione tra i vari componenti del sistema, garantendo coerenza e flessibilità.

Un esempio pratico di applicazione è rappresentato dall'ambito sanitario. Ogni reparto di una struttura medica potrebbe disporre di un chatbot addestrato su documenti specifici, come protocolli clinici o linee guida. I chatbot, accessibili tramite un'interfaccia grafica condivisa, consentirebbero ai medici di consultare rapidamente informazioni pertinenti, migliorando la collaborazione interdisciplinare e l'efficienza operativa.

L'integrazione tra i servizi REST e i modelli LLMs offre molteplici vantaggi:

Uno degli aspetti più significativi dell'uso di REST in combinazione con gli LLMs è la capacità di fornire esperienze personalizzate. Analizzando le richieste degli utenti, i sistemi possono adattare le risposte ai bisogni specifici, migliorando non solo la soddisfazione dell'utente ma anche l'efficacia delle interazioni. Questo approccio personalizzato è particolarmente utile in contesti come il customer service, dove è cruciale rispondere rapidamente e accuratamente a un'ampia varietà di richieste.

Inoltre, la natura stateless di REST garantisce interazioni robuste, minimizzando il rischio di errori legati a sessioni instabili o a interruzioni nella comunicazione. Questo aspetto è particolarmente rilevante in applicazioni mission-critical, come quelle mediche o finanziarie, dove l'affidabilità è essenziale. L'architettura REST, combinata con le capacità degli LLMs, consente di bilanciare richieste ad alta intensità computazionale, garantendo prestazioni fluide anche in situazioni di carico elevato.

Un altro vantaggio significativo è rappresentato dalla scalabilità. I sistemi distribuiti basati su REST possono essere facilmente estesi per gestire un numero crescente di richieste, rendendo questa combinazione ideale per applicazioni globali che operano in ambienti multilingue e multi-dominio. Ad esempio, i chatbot possono fornire risposte in diverse lingue, accedendo a fonti di informazione localizzate e garantendo un'esperienza uniforme agli utenti di tutto il mondo.

Un caso emblematico di miglioramento dell'esperienza utente attraverso REST e LLMs è l'implementazione di sistemi di assistenza virtuale in tempo reale. In questi scenari, i chatbot possono integrare informazioni provenienti da database esterni e modelli predittivi per fornire risposte contestuali e aggiornate. L'uso del caching, inoltre, riduce i tempi di latenza, migliorando ulteriormente l'interazione utente-macchina.

Infine, l'adattabilità di questa combinazione è fondamentale per supportare applicazioni in costante evoluzione. Gli sviluppatori possono aggiornare i modelli di linguaggio o modificare le API REST senza interrompere il servizio, garantendo così un ciclo di sviluppo continuo e una rapida risposta alle nuove esigenze degli utenti.

Il capitolo "Analisi e progettazione" approfondisce i legami tra i servizi RESTful e i modelli di linguaggio avanzati, evidenziando come questa combinazione rappresenti una soluzione potente per lo sviluppo di applicazioni moderne. Attraverso un'analisi dettagliata delle scelte tecnologiche e dell'architettura implementata, si dimostra come i principi di REST possano essere sfruttati per creare sistemi scalabili, modulari e altamente efficienti. Questo approccio non solo soddisfa le esigenze attuali, ma offre anche una solida base per futuri sviluppi tecnologici.

4.2 Applicazione

Per testare la fattibilità di integrare chatbot addestrati con la metodologia RAG (Retrieval-Augmented Generation) in un sistema complesso basato su microservizi e chiamate REST, è stato scelto come caso di studio l'ambito sanitario. In particolare, si è sviluppato un sistema per la gestione dei pazienti che include un chatbot addestrato su alcune schede dietetiche tratte dal progetto SISTER. Quest'ultimo progetto, incentrato sull'invecchiamento attivo degli anziani, ha dedicato particolare attenzione alle diete personalizzate come strumento per prevenire la fragilità biopsicosociale, dimostrando come un approccio multidisciplinare possa migliorare la qualità della vita attraverso interventi mirati e basati su evidenze scientifiche.

Si sono creati diversi microservizi in SpringBoot:

- bff
- bot alimentazione
- ms infermiere
- ms medico
- ms pazienti
- system management

Il BFF rappresenta lo strato di comunicazione tra i microservizi e l'interfaccia di frontend, fungendo da intermediario per gestire le richieste e restituire i dati in modo ottimale. Il progetto Bot Alimentazione è dedicato alla gestione del chatbot addestrato sulle schede dietetiche, fornendo risposte personalizzate e basate sui dati nutrizionali. Il microservizio MS Infermiere si occupa della gestione degli infermieri e delle loro attività operative, mentre MS Medico è responsabile delle funzionalità legate ai medici, come la gestione delle consultazioni e delle prescrizioni. Allo stesso modo, MS Pazienti è focalizzato sulle operazioni riguardanti i pazienti, inclusi l'archiviazione dei dati clinici e il monitoraggio delle loro attività. Infine, il progetto System Management è stato progettato per gestire e addestrare i chatbot, garantendo un aggiornamento continuo delle funzionalità e un miglioramento costante delle prestazioni. Questa architettura modulare consente una chiara separazione delle responsabilità, assicurando flessibilità e scalabilità all'intero sistema.

I bot sviluppati in Python sono invece:

- bot alimentazione
- bot sanitario

Il bot dedicato all'alimentazione comunica con il microservizio corrispondente, Bot Alimentazione, sviluppato in Spring Boot. Questo microservizio è addestrato utilizzando file PDF contenenti informazioni relative all'alimentazione, consentendo al bot di fornire risposte accurate e personalizzate in questo ambito. Il Bot Sanitario, invece, interagisce con i microservizi MS Infermiere, MS Medico e MS Pazienti. Questo bot può essere addestrato su materiale sanitario specifico, rendendolo uno

strumento versatile e utile per medici, infermieri e pazienti, fornendo supporto in base alle esigenze di ciascun ruolo.

Tutti i microservizi interagiscono con un database progettato e implementato utilizzando SQL, che funge da archivio centrale per la gestione delle informazioni. Nel database vengono memorizzati i dati relativi all'addestramento dei bot, comprese le cartelle dei documenti caricati, insieme alle tabelle dedicate alla gestione delle sezioni riguardanti medici, infermieri e pazienti. Questo approccio garantisce una struttura organizzata e un accesso efficiente ai dati necessari per il funzionamento e l'evoluzione del sistema.

Ogni bot si basa su queste tecnologie:

- Python
- LangChain
- Ollama

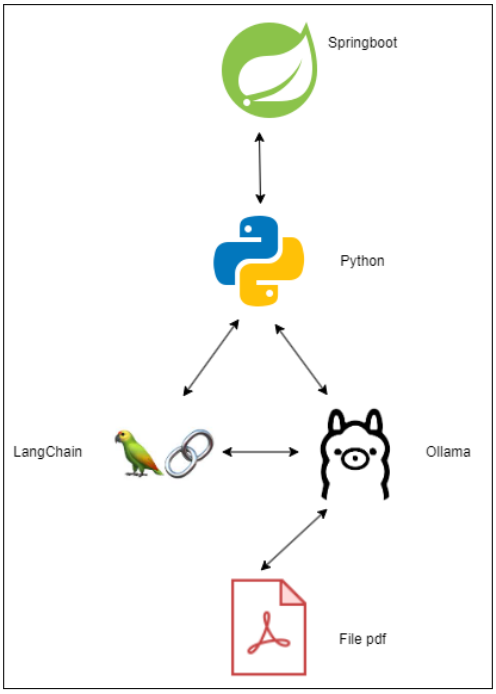
Spring Boot è utilizzato per lo sviluppo dei servizi RESTful che costituiscono il cuore del backend del sistema. Questa potente piattaforma Java semplifica la creazione di applicazioni stand-alone e basate su microservizi. Spring Boot consente di costruire API REST robuste e scalabili, gestendo in modo efficiente le comunicazioni tra il frontend e i vari componenti del backend, e garantendo un'architettura modulare e facilmente estendibile.

Python, infine, è impiegato per la creazione e l'addestramento dei bot. Conosciuto per la sua sintassi chiara e la vasta gamma di librerie specializzate, Python è particolarmente adatto per il machine learning e l'elaborazione del linguaggio naturale. Le librerie di Python, come TensorFlow e PyTorch, permettono di sviluppare modelli avanzati per i bot, che possono essere addestrati per rispondere in modo preciso e contestuale alle esigenze degli utenti.

LangChain è un framework a supporto di Python per la vettorizzazione dei documenti sui quali si desidera addestrare il bot, per la tokenizzazione della domande

dell'utente e il calcolo del coefficente di similarità tra la tokenizzazione della domanda e la tokenizzazione dei documenti. LangChain si coordina con Ollama, software per il download e l'avvio di LLM in locale.

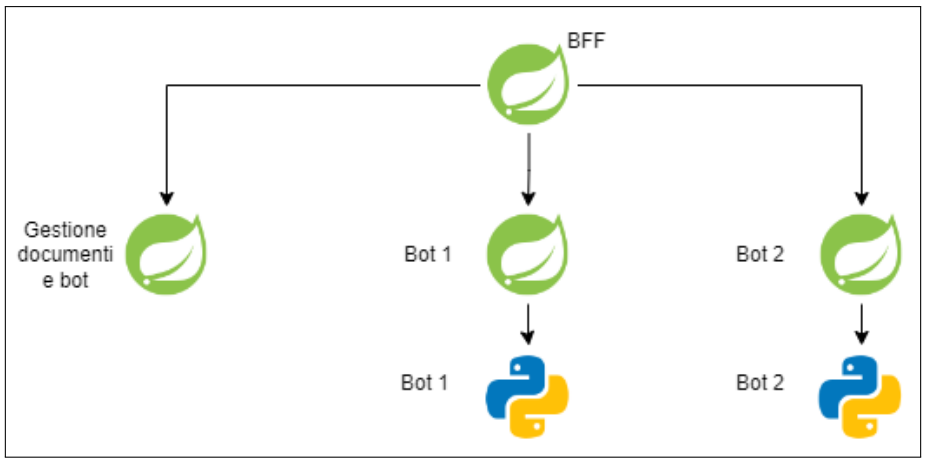
La relazione tra queste tecnologie può essere rappresentata in modo chiaro ed efficace attraverso il seguente schema:



Ollama permette di scaricare e utilizzare localmente il modello linguistico (LLM) desiderato, offrendo una soluzione che coniuga efficienza e sicurezza. Questa tecnologia consente di eseguire l'intero programma direttamente su un server locale, eliminando la necessità di trasferire dati sensibili verso piattaforme cloud esterne. Grazie a questa architettura, è possibile conservare i file PDF e altri dati critici in un ambiente controllato, riducendo al minimo i rischi di accessi non autorizzati o violazioni della privacy.

Oltre a garantire una maggiore protezione dei dati, l'approccio locale contribuisce a ridurre i costi operativi legati all'infrastruttura. L'assenza di dipendenze da servizi cloud esterni non solo abbatte i costi ricorrenti di archiviazione e calcolo, ma aumenta anche l'autonomia nella gestione del sistema, semplificando il controllo delle risorse hardware e software. Questa configurazione è particolarmente vantaggiosa

per aziende e organizzazioni che devono trattare dati sensibili o rispettare rigorosi requisiti di conformità normativa.



In un'infrastruttura complessa composta da più bot che operano contemporaneamente, l'orchestrazione del sistema può essere organizzata in modo chiaro e modulare. Ciascun bot è costituito da due componenti principali: un progetto Python, che si occupa dell'addestramento e dell'interrogazione del modello linguistico (LLM) sui relativi file PDF, e un servizio Spring Boot, che funge da ponte per consentire al bot di comunicare con il resto dell'applicativo.

A supporto di questa struttura, esiste un progetto Spring Boot separato, dedicato alla gestione del caricamento dei documenti per ogni bot. Questo servizio è inoltre responsabile della registrazione dei bot nel database, garantendo un'integrazione coerente e centralizzata delle varie istanze operative.

Per completare l'infrastruttura, un ulteriore progetto Spring Boot è implementato come Backend For Frontend (BFF), che funge da intermediario tra un eventuale frontend e l'intero backend. Questa soluzione facilita le interazioni, offrendo un punto di accesso unico e semplificando la comunicazione tra le varie componenti del sistema.

Questa configurazione modulare non solo rende l'infrastruttura facilmente scalabile, consentendo l'aggiunta di nuovi bot, ma garantisce anche una gestione efficiente e una comunicazione fluida tra le diverse parti dell'ecosistema.

4.2.1 Creazione e addestramento dei chatbot

Nell'infrastruttura creata, ogni volta che è necessario sviluppare e addestrare un nuovo chatbot su documenti PDF specifici di una determinata tematica, viene creato un progetto in Python che utilizza quattro librerie chiave:

- Flask: Questa libreria consente di creare endpoint REST, facilitando la comunicazione tra il chatbot e l'applicazione backend basata su Spring Boot tramite il protocollo REST.
- LangChain e Chroma: Queste librerie sono responsabili della vettorizzazione dei documenti PDF e del recupero delle informazioni rilevanti in risposta ai prompt inseriti dagli utenti attraverso l'interfaccia grafica.
- Ollama: Permette l'interazione tra il programma e il modello di LLM (Large Language Model) installato localmente, gestendo le richieste e le risposte del chatbot.

Parallelamente, viene avviato un progetto in Spring Boot per gestire la comunicazione tra il chatbot e l'interfaccia grafica tramite il protocollo REST, garantendo un'integrazione fluida e coerente tra il frontend e il backend.

```
@app.route('/message', methods=['POST'])
def botAlimentazioneMessage():
    jsonContent = request.json
    query = jsonContent.get('query')
    print(f"Query: {query}")

    response = llm.invoke(query)
    responseAnswer = {"message": check_and_translate(response)}

    return responseAnswer


@app.route('/load-pdf', methods=['POST'])
def loadPdf():
    file = request.files['file']
    fileName = file.filename
    saveFile = ""
```

```
    if os.name == 'nt':
        saveFile = pathAddestramento + "\\" + fileName
    else:
        saveFile = pathAddestramento + "/" + fileName
    file.save(saveFile)
    print(f"File salvato: {saveFile}")

    if not fileName[-4:] == ".pdf":
        response = {
            "status": "success",
            "fileName": fileName,
            "docLen": 0,
            "chunks": 0
        }
        return response

    loadPdf = PDFPlumberLoader(saveFile)

    docs = loadPdf.load_and_split()
    print(f"Doc len: {len(docs)}")

    chunks = text_splitter.split_documents(docs)
    print(f"Doc len: {len(chunks)}")

    response = {
        "status": "success",
        "fileName": fileName,
        "docLen": len(docs),
        "chunks": len(chunks)
    }
    if(len(docs)==0 or len(chunks)==0 ):
        response = {
            "status": "failed",
            "fileName": fileName,
            "docLen": len(docs),
            "chunks": len(chunks)
        }
    else:
        vectorStore = Chroma.from_documents(documents=chunks,
embedding=embedding, persist_directory=pathAddestramento)
```

```
57          vectorStore.persist()
58      return response
59
60
61  @app.route('/message-pdf', methods=['POST'])
62  def askPdf():
63      jsonContent = request.json
64      query = jsonContent.get('query')
65      print(f"Query: {query}")
66
67      print(f"Carico il VectorStore")
68      vectorStore = Chroma(persist_directory=pathAddestramento,
69                           embedding_function=embedding)
70
71      print(f"Creo la chain")
72      retriever = vectorStore.as_retriever(
73          search_type="similarity_score_threshold",
74          search_kwargs={
75              "k": 20,
76              "score_threshold": 0.3,
77          },
78      )
79
80      document_chain = create_stuff_documents_chain(llm, rawPrompt)
81      chain = create_retrieval_chain(retriever, document_chain)
82
83      result = chain.invoke({"input": f"Rispondi in italiano: {query}
    "})
84
85      print(result)
86
87      sources = []
88      for doc in result["context"]:
89          sources.append(
90              {"source": doc.metadata["source"], "pageContent": doc.
    page_content}
91          )
92
93      responseAnswer = {"answer": check_and_translate(result["answer"
    ]), "sources": sources}
```

```
94
95      return responseAnswer
96
97
98  def startApplication():
99      connessioneDb()
100     recuperoPathAddestramento()
101     app.run(host='127.0.0.1', port=5002, debug=True)
```

Listing 4.1: Microservizi Python

Ogni bot sarà collegato a una porta specifica e avrà tre endpoint in Python:

- "/load-pdf": per caricare il file e addestrare il bot sul file pdf
- "/message": per dialogare con LLM non addestrato
- "/message-pdf": per dialogare con LLM addestrato

Quando viene richiamato il path "/load-pdf", il file caricato viene tokenizzato e memorizzato all'interno del vectorstore. Successivamente, richiamando l'endpoint "/message-pdf", l'algoritmo tokenizza la domanda dell'utente e la confronta con i token dei file PDF già presenti nel vectorstore per fornire una risposta pertinente.

```
@Operation(summary = "Chat normale",
            description = "Invio di un messaggio al bot
    Alimentazione sfruttando l'LLM non addestrato")
     @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "
    Operazione andata a buon fine"),
            @ApiResponse(responseCode = "500", description = "
    Errore di sistema")
     })
     @GetMapping(value = "/normal-chat", produces = MediaType.
    APPLICATION_JSON_VALUE)
     public ResponseEntity<GenericResponseDto<String>> normalChat(
    @RequestParam String messagge) {
         return ResponseEntity.ok(esitoMessaggiRequestContextHolder.
    buildGenericResponse(chatService.normalChat(messagge)));
     }

     @Operation(summary = "Chat addestrata",
            description = "Invio di un messaggio al bot
    Alimentazione sfruttando l'LLM addestrato")
     @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "
    Operazione andata a buon fine"),
            @ApiResponse(responseCode = "500", description = "
    Errore di sistema")
     })
     @GetMapping(value = "/chat-addestrata", produces = MediaType.
    APPLICATION_JSON_VALUE)
     public ResponseEntity<GenericResponseDto<ResponseMessagePdfDto
    >> chatAddestrata(@RequestParam String messagge) {
         return ResponseEntity.ok(esitoMessaggiRequestContextHolder.
    buildGenericResponse(chatService.chatAddestrata(messagge)));
     }
```

Listing 4.2: Microservizio Java

Nel corrispettivo progetto in Java, abbiamo i seguenti endpoint:

- "/normal-chat"
- "/chat-addestrata"

Il primo endpoint consente di inviare un messaggio all'applicazione per ottenere una risposta generata da un modello di linguaggio non addestrato. Quando un utente invia un messaggio a questo endpoint, il sistema lo inoltra a un servizio Python che interroga il modello AI non personalizzato. La risposta generata viene quindi restituita all'utente in modo diretto, senza ulteriori informazioni di supporto. Il secondo endpoint, offre un'interazione più sofisticata. Anche in questo caso, l'utente fornisce un messaggio, ma il sistema lo inoltra a un servizio Python diverso, che interroga un modello di linguaggio addestrato su un insieme specifico di dati. Questo modello non solo genera una risposta, ma restituisce anche i file e le sezioni di contenuto utilizzate per formulare la risposta. Ciò consente all'utente di comprendere su quali informazioni il modello ha basato la sua elaborazione, rendendo questa opzione particolarmente utile per casi d'uso che richiedono maggiore trasparenza e tracciabilità.

Questi due endpoint rappresentano quindi due livelli distinti di interazione con i modelli di linguaggio: uno generico e immediato, l'altro personalizzato e più informativo.

4.2.2 Progettazione del database

Durante la fase di sviluppo è stato utilizzato un database relazionale, gestito tramite il linguaggio SQL, denominato botRag. Per configurare lo schema di base necessario al funzionamento dei microservizi, è sufficiente eseguire lo script creazioneTabelle.sql, che contiene tutte le query necessarie per la creazione delle tabelle. Questo script definisce la struttura del database, garantendo che tutti i microservizi possano accedere e manipolare i dati in modo corretto.

L'URL di connessione al database utilizzato è: jdbc:mysql://localhost:3306/. Nel caso in cui la porta di connessione venga modificata o si decida di utilizzare un database in cloud, sarà necessario aggiornare il file di configurazione nei vari microservizi e nei progetti Python. Questo passaggio è fondamentale per garantire il corretto collegamento al database e il funzionamento del sistema.

Il sistema di gestione dei dati per l'addestramento dei bot è organizzato secondo un modello relazionale composto da tre tabelle principali: **rag_bot_pdf**, **cartelle** e **documenti**. Di seguito viene descritto il ruolo e la struttura di ciascuna tabella, nonché le relazioni tra di esse. La tabella **rag_bot_pdf** contiene le configurazioni necessarie per ciascun bot. Qui di seguito vengono descritti i campi della tabella:

- `id_rag_bot_pdf`: Identificativo univoco del bot.
- `nome_bot`: nome del bot
- `descrizione_bot`: descrizione del bot
- `url_spring`: URL utilizzato per l'integrazione con servizi esterni implementati in Spring.
- `url_python`: URL utilizzato per l'integrazione con servizi esterni implementati in Python
- `id_cartella_addestramento`: Riferimento alla cartella che contiene i file utilizzati per l'addestramento singolo del bot.
- `id_cartella_caricamento_massivo`: Riferimento alla cartella utilizzata per la funzionalità di addestramento massivo.

La tabella **cartelle** è destinata alla gestione delle directory utilizzate per l'addestramento. Qui di seguito vengono descritti i campi della tabella:

- `id_cartella`: Identificativo univoco della cartella.
- `nome_cartella`: Nome della cartella.
- `path_cartella`: Percorso della cartella nel file system.

- `is_cartella_addestramento`: Campo booleano che distingue tra:
  - Cartella di addestramento singolo: Contiene i file già pronti per essere utilizzati nel processo di addestramento del bot.
  - Cartella di addestramento massivo: Utilizzata esclusivamente per conservare file che verranno successivamente spostati nella cartella di addestramento singolo al momento dell'esecuzione della funzionalità di addestramento massivo.

Il campo `is_cartella_addestramento` gioca un ruolo fondamentale nel distinguere le due tipologie di cartelle. La cartella di addestramento singolo è utilizzata direttamente nel processo di addestramento del bot, mentre la cartella di addestramento massivo funge da spazio temporaneo per conservare file che verranno successivamente caricati nella cartella di addestramento singolo. Questa distinzione consente di gestire in modo efficiente processi di addestramento su larga scala, garantendo la separazione tra i file temporanei e quelli effettivamente utilizzati dal bot.

La tabella documenti rappresenta i file utilizzati nel processo di addestramento. Qui di seguito vengono descritti i campi della tabella:

- `id_documento`: Identificativo univoco del documento.
- `nome_documento`: Nome del file.
- `estensione_documento`: Estensione o formato del file
- `path_documento`: Percorso del file nel file system.
- `id_cartella`: Collegamento alla cartella di appartenenza.

Le tre tabelle sono collegate per formare un sistema relazionale:

- La tabella `rag_bot_pdf` è associata alla tabella cartelle tramite i campi `id_cartella_addestramento` e `id_cartella_caricamento_massivo`, consentendo di associare un bot a due diverse directory: una per l'addestramento singolo e una per il caricamento massivo.

- La tabella cartelle, a sua volta, è collegata alla tabella documenti tramite il campo `id_cartella`, che associa ogni documento a una specifica cartella.

Le altre tabelle del database sono destinate alla gestione del sistema sanitario in cui l'applicativo sarà utilizzato. Non fanno parte della gestione dei bot, ma possono essere adattate in base al contesto in cui i bot vengono creati. In futuro, si prevede comunque la possibilità di addestrare un bot utilizzando i dati presenti nel database e di sviluppare una memoria interna per il bot.

4.2.3 Creazione dei microservizi REST

Dopo aver creato i bot attraverso i relativi progetti in Python, come descritto nella sezione 2.1.1, si è proceduto allo sviluppo dei corrispondenti microservizi utilizzando Spring Boot, inclusa la realizzazione del microservizio System Management. I progetti relativi ai bot espongono endpoint specifici per consentire la comunicazione con gli altri componenti del sistema.

Il microservizio System Management adotta la metodologia Object-Relational Mapping (ORM) per mappare le entità del database su oggetti Java, agevolando la manipolazione dei dati e garantendo una stretta integrazione con la logica applicativa. Questo microservizio fornisce endpoint per operazioni essenziali come la registrazione, l'aggiornamento e l'eliminazione dei bot. Inoltre, include funzionalità per il caricamento dei file necessari e l'addestramento dei modelli, contribuendo a una gestione centralizzata ed efficace del ciclo di vita dei bot.

Un ruolo cruciale è svolto dal microservizio BFF (Backend for Frontend), progettato per fungere da ponte tra il front-end e gli altri microservizi. Al suo interno vengono memorizzati i file YAML, che descrivono le configurazioni e le specifiche di comunicazione tra i vari microservizi. Questo approccio semplifica il flusso dei dati, migliorando la modularità e riducendo la complessità nelle interazioni tra front-end e back-end. Il BFF consente al sistema di mantenere una chiara separazione delle responsabilità: i microservizi relativi ai bot e al System Management gestiscono rispettivamente le funzionalità core e la logica applicativa, mentre il BFF offre un'interfaccia ottimizzata per il front-end, migliorando l'esperienza utente e l'efficienza complessiva del sistema.

La configurazione dei path necessari per la comunicazione tra i microservizi è gestita dalla classe ExternalApiClientConfig, che centralizza e organizza i riferimenti ai percorsi esposti. Questo approccio garantisce una struttura modulare e facilmente manutenibile, permettendo di stabilire in modo chiaro e flessibile i percorsi per invocare le funzionalità offerte dai microservizi.

La gestione delle risposte derivanti dalle chiamate ai microservizi è affidata alla classe EsitoResponseErrorHandler, che si occupa di convertire e uniformare le risposte ricevute. Tali risposte vengono incapsulate in un oggetto generalizzato del tipo generico T, che include:

- un identificativo dell'operazione
- una lista di messaggi che possono contenere informazioni, avvisi o errori,
- il tipo di esito che specifica se l'operazione è andata a buon fine o meno.

Questa struttura consente una gestione uniforme delle risposte, migliorando la leggibilità e la manutenzione del codice. La generalizzazione introdotta dalla classe EsitoResponseErrorHandler offre inoltre maggiore flessibilità nel trattamento delle risposte, indipendentemente dal tipo specifico di payload restituito.

Per facilitare l'interazione con gli endpoint esposti dai microservizi, è stato integrato Swagger UI, un'interfaccia grafica intuitiva che permette agli sviluppatori e agli utenti di esplorare e testare gli endpoint. Swagger UI genera automaticamente documentazione interattiva basata su annotazioni presenti nel codice, rendendo l'accesso agli endpoint più immediato e agevole. Questa integrazione offre una soluzione user-friendly per inviare richieste e visualizzare risposte, eliminando la necessità di strumenti esterni o conoscenze approfondite dei dettagli di implementazione.

Capitolo 5

Risultati

Risultati ed esempio di risposta

Capitolo 6

Conclusione

riepilogo di tutto quello che hai fatto e commenta i risultati ottenuti in fase di sperimentazione